МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Автоматизированные системы управления»

# ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ

***Методические рекомендации к лабораторным работам для студентов специальности 6-05-0611-01 «Информационные системы и технологии» очной и заочной форм обучения***


Могилев 2026

УДК 004.42
ББК 32.973-018
П78

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Автоматизированные системы управления» «18» ноября 2025 г., протокол № 4

Составитель канд. техн. наук, доц. И. В. Акиншева

Рецензент канд. техн. наук, доц. Ю. С. Романович

Методические рекомендации к лабораторным работам по дисциплине «Программирование сетевых приложений» предназначены для студентов специальности 6-05-0611-01 «Информационные системы и технологии» очной и заочной форм обучения. Приведены задания и список литературы для подготовки.

Учебное издание

ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ

| | |
|---|---|
| Ответственный за выпуск | А. И. Якимов |
| Корректор | И. В. Голубцова |
| Компьютерная верстка | Н. П. Полевничая |

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 21 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

© Белорусско-Российский
университет, 2026

# Содержание

# Введение

Подготовка современного специалиста требует уверенного владения возможностями, предоставляемыми компьютерными технологиями. Изучение настоящей учебной дисциплины обеспечивает подготовку специалиста, владеющего фундаментальными знаниями и практическими навыками в области основ программирования сетевых приложений при решении практических задач.

Цель учебной дисциплины состоит в подготовке специалистов, владеющих базовыми знаниями, умениями и практическими навыками в области языков и средств разработки сетевых приложений, ориентированных на клиент-серверную архитектуру, программирования элементов такой архитектуры.

Методические рекомендации предназначены для помощи студентам в выполнении лабораторных работ по дисциплине.

В методических рекомендациях для изучения представлены сведения о реализации многопоточности, апплетах, интерфейсе JDBC, сетевом программировании с сокетами и каналами, установлении соединения с разработанной базой данных, удаленном вызове метода (RMI).

При выполнении лабораторных работ используется следующее программное обеспечение: операционная система Microsoft Windows; пакет jdk; среды разработки JCreatorPro или NetBeans IDE, или Eclipse, или IntelliJ IDEA; web-сервера Tom Cat или Glass Fish; СУБД MySQL, клиент для MySQL – HeidiSQL или инструмент для визуального проектирования баз данных MySQL Workbench.

# 1 Лабораторная работа № 1. Потоки выполнения

**Цель работы**: изучить основные способы реализации многопоточности в Java.

**Порядок выполнения работы.**

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.

3 Оформить отчет.

**Требования к отчету.**

1 Цель работы.

2 Полученные результаты.

3 Выводы.

***Теоретические сведения***

**Реализация многопоточности в Java.**

Для реализации многопоточности мы должны воспользоваться классом java.lang.Thread. В этом классе определены все методы, необходимые для создания потоков, управления их состоянием и синхронизации.

Как пользоваться классом Thread? Есть две возможности.

Во-первых, вы можете создать свой дочерний класс на базе класса Thread. При этом вы должны переопределить метод run. Ваша реализация этого метода будет работать в рамках отдельного потока.

Во-вторых, ваш класс может реализовать интерфейс Runnable. При этом в рамках вашего класса необходимо определить метод run, который будет работать как отдельный поток.

Второй способ особенно удобен в тех случаях, когда ваш класс должен быть унаследован от какого-либо другого класса и при этом вам нужна многопоточность. Так как в языке программирования Java нет множественного наследования, невозможно создать класс, для которого в качестве родительского будут выступать классы Applet и Thread. В этом случае реализация интерфейса Runnable является единственным способом решения задачи.

**Методы класса Thread.**

В классе Thread определены три поля, несколько конструкторов и большое количество методов, предназначенных для работы с потоками. Ниже приведено краткое описание полей, конструкторов и методов.

С помощью конструкторов вы можете создавать потоки различными способами, указывая при необходимости для них имя и группу. Имя предназначено для идентификации потока и является необязательным атрибутом. Что же касается групп, то они предназначены для организации защиты потоков друг от друга в рамках одного приложения.

Методы класса Thread предоставляют все необходимые возможности для управления потоками, в том числе для их синхронизации.

**Реализация интерфейса Runnable.**

Описанный выше способ создания потоков как объектов класса Thread или унаследованных от него классов кажется достаточно естественным. Однако этот способ не единственный. Если вам нужно создать только один поток, работающий одновременно с кодом Applet, то проще выбрать второй способ с использованием интерфейса Runnable.

Идея заключается в том, что основной класс апплета, который является дочерним по отношению к классу Applet, дополнительно реализует интерфейс Runnable, как это показано ниже:

```
public class MultiTask extends Applet implements Runnable
{
Thread m_MultiTask = null;
. . .
public void run()
{
. . .
}
public void start()
{
if (m_MultiTask == null)
{
m_MultiTask = new Thread(this); m_MultiTask.start();
}
}
public void stop()
{
if (m_MultiTask != null)
{
m_MultiTask.stop(); m_MultiTask = null;
}
}
}
```

Внутри класса необходимо определить метод run, который будет выполняться в рамках отдельного потока. При этом можно считать, что код апплета и код метода run работают одновременно как разные потоки.

Для создания потока используется оператор new. Поток создается как объект класса Thread, причем конструктору передается ссылка на класс апплета:

```
m_MultiTask = new Thread(this);
```

При этом, когда поток запустится, управление получит метод run, определенный в классе Applet. Как запустить поток? Запуск выполняется, как и раньше, методом start. Обычно поток запускается из метода start Applet, когда поль-

зователь отображает страницу сервера Web, содержащую Applet. Остановка потока выполняется методом stop.

**Завершение и остановка потоков.**

Как уже указывалось, нормальное завершение нити происходит при выходе из метода run. Кроме того, иногда в приложении требуется выполнить временную остановку нити, чтобы потом возобновить ее работу.

В классе Thread есть методы stop (завершить), suspend (приостановить) и resume (возобновить), но все они объявлены deprecated (устаревшими) и их использование не рекомендовано. О причинах можно почитать подробнее в документации по Java и их здесь рассматривать не будем.

Как же поступить в случае, если нужно приостановить выполнение процесса? Для этого нужно не останавливать процесс, а обеспечить такой алгоритм, при котором он работает вхолостую.

**Приоритеты потоков.**

Если процессор создал несколько потоков, то все они выполняются параллельно, причем время центрального процессора (или нескольких центральных процессоров в мультипроцессорных системах) распределяется между этими потоками.

Распределением времени центрального процессора занимается специальный модуль операционной системы – планировщик. Планировщик по очереди передает управление отдельным потокам, так что даже в однопроцессорной системе создается полная иллюзия параллельной работы запущенных потоков.

Распределение времени выполняется по прерываниям системного таймера. Поэтому каждому потоку дается определенный интервал времени, в течение которого он находится в активном состоянии.

Заметим, что распределение времени выполняется для потоков, а не для процессов. Потоки, созданные разными процессами, конкурируют между собой за получение процессорного времени.

Для оптимизации параллельной работы нитей в Java имеется возможность устанавливать приоритеты нитей. Нити с большим приоритетом имеют преимущество в получении времени процессора перед нитями с более низким приоритетом.

Работа с приоритетами обеспечивается методами класса Thread.

Метод, который устанавливает приоритет потока, – public final void setPriority(int newPriority).

Метод, который позволяет установить приоритет потока, – public final int getPriority().

Значение параметра в методе setPriority не может быть произвольным. Оно должно находиться в пределах от MIN_PRIORITY до MAX_PRIORITY. При своем создании нить имеет приоритет NORM_PRIORITY.

**Средства синхронизации потоков в Java.**

Проанализируем эту проблему с другой точки зрения. У нас есть некоторый ресурс, в данном случае – случайное число (переменная randValue), доступ к которому нужно упорядочить. То есть нужно заблокировать доступ к этому ресурсу для всех нитей, кроме одной, пока эта нить не выполнит над ним необходимые действия.

В Java есть возможности по синхронизации нитей, построенные на этом принципе. То есть определяется некоторый ресурс, который может быть заблокирован. Таким ресурсом может быть любой объект (но не данные элементарного типа). Далее определяются критические участки программы. При входе в такой участок ресурс блокируется и становится недоступным для всех других нитей (с некоторой оговоркой, которую мы рассмотрим позже). При выходе из критического участка выполняется разблокировка ресурса.

В качестве критического участка программы в Java может быть определен некоторый нестатический метод или блок. При вызове метода блокируется объект, для которого вызван данный метод (this). Для блока блокируемый объект указывается явно.

Синтаксис определения критических участков следующий.

Для метода мы просто при описании метода указываем описатель synchronized, например:

```
public void synchronized f() {
...
}
```

Для блока мы непосредственно перед блоком ставим конструкцию synchronized (объект), где объект – это ссылка на блокируемый объект. Например:

```
synchronized(ref) {
...
}
```

Здесь в качестве критического участка определяется блок, а в качестве блокируемого объекта – объект, на который ссылается ref.

**Задания**

1 Создать Applet, используя поток: строка движется горизонтально, отражаясь от границ Applet и меняя при этом случайным образом свой цвет.

2 Создать Applet, используя поток: строка движется по диагонали. При достижении границ Applet все символы строки случайным образом меняют регистр.

3 Организовать сортировку массива методами Шелла, Хоара, пузырька на основе бинарного дерева в разных потоках.

4 Реализовать сортировку графических объектов, используя алгоритмы из задания 3.4.

5 Создать Applet с точкой, движущейся по окружности с постоянной угловой скоростью. Сворачивание браузера должно приводить к изменению угловой скорости движения точки для следующего запуска потока.

6 Изобразить точку, пересекающую с постоянной скоростью окно слева направо (справа налево) параллельно горизонтальной оси. Как только точка доходит до границы окна, в этот момент от левого (правого) края с вертикальной координатой y, выбранной с помощью датчика случайных чисел, начинает свое движение другая точка и т. д. Цвет точки также можно выбирать с помощью датчика случайных чисел. Для каждой точки создается собственный поток.

7 Изобразить в приложении правильные треугольники, вращающиеся в плоскости экрана вокруг своего центра. Каждому объекту соответствует поток с заданным приоритетом.

8 Условия предыдущих задач изменяются таким образом, что центр вращения перемещается от одного края окна до другого с постоянной скоростью параллельно горизонтальной оси.

9 Создать фрейм с тремя шариками, одновременно летающими в окне. С каждым шариком связан свой поток со своим приоритетом.

10 Два изображения выводятся в окно. Затем они постепенно исчезают с различной скоростью в различных потоках (случайным образом выбираются точки изображения и их цвет устанавливается в цвет фона).

11 Условие предыдущей задачи изменить на применение эффекта постепенного проявления двух изображений.

12 Создать Applet «Бегущая строка».

13 Реализовать приложение, в котором пользователь имеет возможность указывать маски файлов для поиска и набор путей, по которым эти файлы нужно искать (например, список логических дисков).

14 Написать секундомер – класс Stopwatch – для замера времени в отдельном потоке выполнения. В классе должны быть реализованы следующие методы:

– start – начинает отсчет времени;

– stop – прерывает отсчет времени;

– reset – сбрасывает текущее значение секундомера;

– getTime – возвращает отсчитанное время в миллисекундах.

Для демонстрации работы секундомера написать консольное приложение.

Пользователю должны быть доступными следующие команды:

– start N – запустить секундомер и дать ему идентификатор N;

– stop N – остановить секундомер с идентификатором N;

– reset N – сбросить время у секундомера с идентификатором N;

– time N – показать время у секундомера с идентификатором N;

– help – список команд;

– exit – выход.

15 Подсчет простых чисел. Написать Swing-приложение для подсчета простых чисел в параллельных потоках. Каждый поток подсчитывает простые числа из заданного диапазона. Всего должно быть запущено три потока подсчета.

Полученные числа выдаются на экран. Можно запускать/останавливать подсчет несколько раз. Пользователю доступны следующие элементы управления:

– кнопка запуска всех потоков;

– кнопка остановки всех потоков;

– поле (JTextArea) для вывода результата в виде

<номер потока>: <число>;

– четыре поля ввода диапазонов чисел для подсчета. Получается три диапазона (между четырьмя числами) для каждого потока;

– три строки с информацией о состоянии каждого потока.

***Контрольные вопросы***

1 Дайте определение понятию «процесс».

2 Дайте определение понятию «поток».

3 Дайте определение понятию «синхронизация потоков».

4 Как взаимодействуют программы, процессы и потоки?

5 В каких случаях целесообразно создавать несколько потоков?

6 Что может произойти, если два потока будут выполнять один и тот же код в программе?

7 Что вы знаете о главном потоке программы?

8 Какие есть способы создания и запуска потоков?

9 Какой метод запускает поток на выполнение?

10 Какой метод описывает действие потока во время выполнения?

11 Когда поток завершает свое выполнение?

# 2 Лабораторная работа № 2. Интерфейс JDBC

**Цель работы**: изучить интерфейс JDBC; научиться создавать приложения с доступом к БД (базе данных).

**Порядок выполнения работы.**

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.

3 Оформить отчет.

**Требования к отчету.**

1 Цель работы.

2 Результаты работы.

3 Выводы.

***Теоретические сведения***

JDBC (Java DataBase Connectivity) – стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и системой управления базами данных (СУБД). Взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов [1].

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным БД, для которых существуют драйверы, предоставляющие способ общения с реальным сервером базы данных [1, 2].

Далее приводится последовательность действий для выполнения первого запроса.

Подключение библиотеки с классом-драйвером БД.

Дополнительно требуется подключить к проекту библиотеку, содержащую драйвер, предварительно поместив ее в папку /lib приложения:

mysql-connector-java-[номер версии]-bin.jar.

**Установка соединения с БД.**

Для установки соединения с БД вызывается статический метод getConnection() класса java.sql.DriverManager. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Загрузка класса драйвера базы данных при отсутствии ссылки на экземпляр этого класса в JDBC 4.0 происходит автоматически при установке соединения экземпляром DriverManager. Метод возвращает объект Connection. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов:

```
Connection cn = DriverManager.getConnection("jdbc:mysql://localhost:3306/testphones", "root", "pass");
```

В результате возвращен объект Connection и имеет место одно установленное соединение с БД с именем testphones. Класс DriverManager предоставляет средства для управления набором драйверов баз данных. С помощью метода getDrivers() можно получить список всех доступных драйверов. До появления JDBC 4.0 объект драйвера СУБД нужно было создавать явно с помощью вызова

```
Class.forName("com.mysql.jdbc.Driver"); или регистрировать драйвер
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

В большинстве случаев в этом нет необходимости, т. к. экземпляр драйвера загружается автоматически при попытке получения соединения с БД [2].

**Создание объекта для передачи запросов.**

После создания объекта Connection и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект Statement, создаваемый вызовом метода createStatement() класса Connection:

```
Statement st = cn.createStatement();
```

Объект класса Statement используется для выполнения SQL-запроса без его предварительной подготовки. Могут применяться также объекты классов PreparedStatement и CallableStatement для выполнения подготовленных запросов и хранимых процедур [2, 3].

**Выполнение запроса.**

Созданные объекты можно использовать для выполнения запроса SQL, передавая его в один из методов: execute(String sql), executeBatch(), executeQuery(String sql) или executeUpdate(String sql). Результаты выполнения запроса помещаются в объект ResultSet:

```
/* выборка всех данных таблицы phonebook */
ResultSet rs = st.executeQuery("SELECT * FROM phonebook");
```

Для добавления, удаления или изменения информации в таблице строка запроса помещается в метод executeUpdate().

Обработка результатов выполнения запроса производится методами интерфейса ResultSet, самыми распространенными из которых являются next(), first(), previous(), last(), группа методов по доступу к информации вида getString(int pos), а также аналогичные методы, начинающиеся с getТип(int pos)(getInt(int pos), getFloat(int pos) и др.) и updateТип(). Среди них следует выделить методы getClob(int pos) и getBlob(int pos), позволяющие извлекать из полей таблицы специфические объекты (Character Large Object, Binary Large Object), которые могут быть, например, графическими или архивными файлами. Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его имени в строке результатов методами типа int getInt(String columnLabel), String getString(String columnLabel), Object getObject(String columnLabel) и подобными им. Интерфейс располагает большим числом методов по доступу к таблице результатов, поэтому рекомендуется изучить его достаточно тщательно. При первом вызове метода next() указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение false [2, 4].

```
Закрытие соединения statemen:
st.close(); // закрывает также и ResultSet cn.close();
```

После того как база больше не нужна, соединение закрывается. Для того чтобы правильно пользоваться приведенными методами, программисту требу-

ется знать типы полей БД. В распределенных системах это знание предполагается изначально. В Java 7 для объектов-ресурсов, требующих закрытия, реализована технология try with resources.

Рассмотрим пример использование СУБД MySQL. СУБД MySQL совместима с JDBC и будет применяться для создания учебных баз данных. Версия СУБД может быть загружена с сайта www.mysql.com. Для корректной установки необходимо следовать инструкциям мастера. В процессе установки следует создать администратора СУБД с именем root и паролем, например, pass. Если планируется разворачивать реально работающее приложение, необходимо исключить тривиальных пользователей сервера БД (иначе злоумышленники могут получить полный доступ к БД). Для запуска следует использовать команду из папки /mysql/bin:

```
mysqld-nt–standalone
```

Если не появится сообщение об ошибке, то СУБД MySQL запущена. Для создания базы данных и ее таблиц используются команды языка SQL [2, 4].

**Создание простого соединения и простого запроса.**

Воспользуемся всеми предыдущими инструкциями и создадим пользовательскую БД с именем testphones и одной таблицей PHONEBOOK. Таблица должна содержать три поля: числовое (первичный ключ) IDPHONEBOOK, символьное LASTNAME, числовое PHONE и несколько занесенных записей (таблица 2.1).

Таблица 2.1 – Пример таблицы из БД

| IDPHONEBOOK | LASTNAME | PHONE |
|---|---|---|
| 1 | Иванов | 9379992 |
| 2 | Петров | 8415141 |

При создании таблицы следует задавать кодировку UTF-8, поддерживающую хранение символов кириллицы. Приложение, осуществляющее простейший запрос на выбор всей информации из таблицы, выглядит следующим образом:

```
import java.sql.Connection; import java.sql.DriverManager; import java.sql.Statement;
import java.sql.ResultSet; import java.sql.SQLException; import java.util.ArrayList; import
java.util.Properties; import data.subject.Abonent;
public class SimpleJDBCRunner { public static void main(String[ ] args) {
String url = "jdbc:mysql://localhost:3306/testphones"; Properties prop = new Proper-
ties();
prop.put("user", "root");
prop.put("password", "pass"); prop.put("autoReconnect", "true");
prop.put("characterEncoding", "UTF-8"); prop.put("useUnicode", "true"); Connection cn =
null;
DriverManager.registerDriver(new com.mysql.jdbc.Driver()); try { // 1 блок
cn = DriverManager.getConnection(url, prop); Statement st = null;
```

```
try { // 2 блок
st = cn.createStatement(); ResultSet rs = null;
try { // 3 блок
rs = st.executeQuery("SELECT * FROM phone- book");
ArrayList lst = new ArrayList<>(); while (rs.next()) {

int id = rs.getInt(1);
int phone = rs.getInt(3); String name = rs.getString(2);
lst.add(new Abonent(id, phone, name));
}
if (lst.size() > 0) { System.out.println(lst);
} else {
System.out.println("Not found");
}
} finally { // для 3-го блока try
/*
* закрыть ResultSet, если он был открыт
* или ошибка произошла во время
* чтения из него данных
*/
if (rs != null) { // был ли создан ResultSet rs.close();
} else { System.err.println(
"ошибка во время чтения из БД");
}
}
} finally {
/*
* закрыть Statement, если он был открыт или если
* произошла ошибка во время создания Statement
*/
if (st != null) { // для 2-го блока try st.close();
} else {
System.err.println("Statement не создан");
}
}
} catch (SQLException e) { // для 1-го блока try System.err.println("DB connection
error: " + e);
/*
* вывод сообщения о всех SQLException
*/
} finally {
/*
* закрыть Connection, если он был открыт
*/

if (cn != null) {
try {
cn.close();
} catch (SQLException e) { System.err.println("Connection close error: " + e);
}
}
}
```

```
}
}
```

В несложном приложении достаточно контролировать закрытие соединения, т. к. незакрытое или «провисшее» соединение снижает быстродействие системы. Класс Abonent, используемый приложением для хранения информации, извлеченной из БД, выглядит очень просто:

```
import java.io.Serializable;
public abstract class Entity implements Serializable, Cloneable { private int id;
public Entity() {
}
public Entity(int id) { this.id = id;
}
public int getId() {
return id;
}
public void setId(int id) { this.id = id;
}
}
public class Abonent extends Entity { private int phone;
private String lastname; public Abonent() {
}
public Abonent(int id, int phone, String lastname) { super(id);
this.phone = phone; this.lastname = lastname;
}
public int getPhone() {

return phone;
}
public void setPhone(int phone) { this.phone = phone;
}
public String getLastname() { return lastname;
}
public void setLastname(String lastname) { this.lastname = lastname;
}
@Override
public String toString() {
return "Abonent [id=" + id + ", phone=" + phone + ", lastname=" + last- name + "]";
}
}
```

Параметры соединения можно задавать несколькими способами: с помощью прямой передачи значений в коде класса, а также с помощью файлов properties или xml. Окончательный выбор производится в зависимости от конфигурации проекта. Чтение параметров соединения с базой данных и получение соединения следует вынести в отдельный класс. Класс ConnectorDB использует файл ресурсов database.properties, в котором хранятся, как правило, параметры подключения к БД, такие как логин и пароль доступа.

Например:

db.driver = com.mysql.jdbc.Driver db.user = root
db.password = pass db.poolsize = 32
db.url = jdbc:mysql://localhost:3306/testphones db.useUnicode = true
db.encoding = UTF-8
Код класса ConnectorDB выглядит следующим образом: import java.sql.Connection;
import java.sql.DriverManager; import java.sql.SQLException; import java.util.ResourceBundle; public class ConnectorDB {
public static Connection getConnection() throws SQLException {

ResourceBundle resource = ResourceBundle.getBundle("database"); String url = resource.getString("db.url");
String user = resource.getString("db.user"); String pass = resource.getString("db.password");
return DriverManager.getConnection(url, user, pass);
}
}

В таком случае получение соединения с БД сведется к вызову Connection cn = ConnectorDB.getConnection();

Метаданные

Существует целый ряд методов интерфейсов ResultSetMetaData и DatabaseMetaData для интроспекции объектов. С помощью этих методов можно получить список таблиц, определить типы, свойства и количество столбцов БД [4]. Для строк подобных методов нет. Получить объект ResultSetMetaData можно следующим образом:

ResultSetMetaData rsMetaData = rs.getMetaData(); Некоторые методы интерфейса ResultSetMetaData:

– int getColumnCount() – возвращает число столбцов набора результатов объекта ResultSet;

– String getColumnName(int column) – возвращает имя указанного столбца объекта ResultSet;

– int getColumnType(int column) – возвращает тип данных указанного столбца объекта ResultSet и т. д.

Получить объект DatabaseMetaData можно следующим образом: DatabaseMetaData dbMetaData = cn.getMetaData();

Некоторые методы весьма обширного интерфейса DatabaseMetaData:

– String getDatabaseProductName() – возвращает название СУБД;

– String getDatabaseProductVersion() – возвращает номер версии СУБД;

– String getDriverName() – возвращает имя драйвера JDBC;

– String getUserName() – возвращает имя пользователя БД;

– String getURL() – возвращает местонахождение источника данных;

– ResultSet getTables() – возвращает набор типов таблиц, доступных для данной БД, и т. д.

**Подготовленные запросы и хранимые процедуры.**

Для представления запросов существует еще два типа объектов: PreparedStatement и CallableStatement.

Объекты первого типа используются при выполнении часто повторяющихся запросов SQL. Такой оператор предварительно готовится и хранится в объекте, что ускоряет обмен информацией с базой данных при многократном выполнении однотипных запросов.

Второй интерфейс используется для выполнения хранимых процедур, созданных средствами самой СУБД.

При использовании PreparedStatement невозможен sql injection attacks. То есть если существует возможность передачи в запрос информации в виде строки, то следует использовать для выполнения такого запроса объект PreparedStatement. Для подготовки SQL-запроса, в котором отсутствуют конкретные параметры, используется метод prepareStatement(String sql) интерфейса Connection, возвращающий объект PreparedStatement [4]:

```
String sql =
"INSERT INTO phonebook(idphonebook, lastname, phone) VALUES(?, ?, ?)"; PreparedStatement ps = cn.prepareStatement(sql);
```

Установка входных значений конкретных параметров этого объекта производится с помощью методов setString(int index, String x), setInt(int index, int x) и подобных им, после чего и осуществляется непосредственное выполнение запроса методами int executeUpdate(), ResultSet executeQuery().

Интерфейс CallableStatement расширяет возможности интерфейса PreparedStatement и обеспечивает выполнение хранимых процедур.

Хранимая процедура – это в общем случае именованная последовательность команд SQL, рассматриваемая как единое целое и выполняющаяся в адресном пространстве процессов СУБД, которые можно вызвать извне (в зависимости от политики доступа используемой СУБД). В данном случае хранимая процедура будет рассматриваться в более узком смысле – как последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных [2–4]. Для создания объекта CallableStatement вызывается метод prepareCall() объекта Connection.

Интерфейс CallableStatement позволяет исполнять хранимые процедуры, которые находятся непосредственно в БД. Одна из особенностей этого процесса заключается в том, что CallableStatement способен обрабатывать не только входные (IN), но и выходные (OUT) и смешанные (INOUT) параметры. Тип выходного параметра должен быть зарегистрирован с помощью метода registerOutParameter(). После установки входных и выходных параметров вызываются методы execute(), executeQuery() или executeUpdate().

Пусть в БД существует хранимая процедура getlastname, которая по уникальному номеру телефона для каждой записи в таблице phonebook будет возвращать соответствующее ему имя:

```
CREATE PROCEDURE getlastname (p_phone IN INT, p_lastname OUT VARCHAR)
AS BEGIN
SELECT lastname INTO p_lastname FROM phonebook WHERE phone = p_phone;
END
```

Тогда для получения имени через вызов данной процедуры необходимо исполнить java-код следующего вида:

```
final String SQL = "{call getlastname (?, ?)}"; CallableStatement cs = cn.prepareCall(SQL);
// передача значения входного параметра cs.setInt(1, 1658468);
// регистрация возвращаемого параметра cs.registerOutParameter(2, java.sql.Types.VARCHAR); cs.execute();
String lastName = cs.getString(2);
```

В JDBC также существует механизм batch-команд, который позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу:

```
// turn off autocommit cn.setAutoCommit(false);
Statement st = con.createStatement();
st.addBatch("INSERT INTO phonebook VALUES (55, 5642032, 'Ивано')"); st.addBatch("INSERT INTO location VALUES (260, 'Minsk')"); st.addBatch("INSERT INTO student_department VALUES (500, 130)");
// submit a batch of update commands for execution int[ ] updateCounts = stmt.executeBatch();
```

Если используется объект PreparedStatement, batch-команда состоит из параметризованного SQL-запроса и ассоциируемого с ним множества параметров. Метод executeBatch() интерфейса PreparedStatement возвращает массив чисел, причем каждое характеризует число строк, которые были изменены конкретным запросом из batch-команды.

Пусть существует список объектов типа Abonent со стандартным набором методов getТип()/setТип() для каждого из его полей и необходимо внести их значения в БД. Многократное использование методов execute() или executeUpdate() становится неэффективным, и в данном случае лучше использовать схему batch-команд:

```
try {
ArrayList abonents = new ArrayList<>(); // заполнение списка PreparedStatement statement = con.prepareStatement("INSERT INTO phone- book VALUES(?,?,?)");
for (Abonent abonent : abonents) { statement.setInt(0, abonent.getId()); statement.setInt(1, abonent.getPhone()); statement.setString(2, abonent.getLastname()); statement.addBatch();
}
updateCounts = statement.executeBatch();
} catch (BatchUpdateException e) { e.printStackTrace();
}
```

**Задание**

Используя интерфейс JDBC, совместно с СУБД MySQL создать приложение согласно варианту.

1 Приложение «Справочник телефонных номеров».
2 Приложение для учета товаров на складе.
3 Приложение для учета билетов в кинотеатре.
4 Приложение для учета билетов автовокзала.
5 Приложение учета заявок по ремонту бытовой техники.
6 Приложение учета заказов в интернет-магазине.
7 Приложение для учета успеваемости студентов.
8 Приложение «Электронный отдел кадров».
9 Приложение «Расписание движения поездов».
10 Приложение для учета оказываемых предприятием услуг.
11 Приложение для учета книг в библиотеке.
12 Приложение для учета курсов валют.
13 Приложение для учета футбольных матчей.
14 Приложение для учета выигрышных лотерейных билетов.
15 Приложение для учета результатов тестирования.

***Контрольные вопросы***

1 Что такое JDBC?
2 Что такое драйвер базы данных?
3 Как выполнить запрос к базе данных?
4 Что называют параметром запроса?
5 С помощью каких интерфейсов можно получить список таблиц, определить типы, свойства и количество столбцов БД?
6 Какой метод возвращает имя указанного столбца в БД?
7 Какой метод возвращает местонахождение источника данных?
8 Для чего используется интерфейс PreparedStatement?
9 Что такое хранимая процедура?
10 Что позволяет использовать интерфейс CallableStatement?

# 3 Лабораторная работа № 3. Сетевое программирование с сокетами и каналами

**Цель работы**: приобрести навыки проектирования и разработки приложений в архитектуре клиент-сервер.

**Порядок выполнения работы.**

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.
2 Получить задание у преподавателя, выполнить задание.
3 Оформить отчет.

**Требования к отчету.**

1 Цель работы.

2 Результаты работы.

3 Выводы.

***Теоретические сведения***

Клиент-сервер (от англ. Client-server) – вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг (сервисов), называемыми серверами, и заказчиками услуг, называемыми клиентами. Нередко клиенты и серверы взаимодействуют через компьютерную сеть и могут быть как различными физическими устройствами, так и программным обеспечением.

Таким образом, работа сервера состоит в прослушивании соединения, она выполняется с помощью специального объекта, который создается пользователем. Работа клиента состоит в попытке создать соединение с сервером, и это выполняется с помощью специального клиентского объекта. Как только соединение установлено, то и клиентская, и серверная стороны соединения превращаются в потоковый объект ввода/вывода. Таким образом, можно трактовать соединение, как будто пользователь читает и пишет файл. Таким образом, после установки соединения используются команды ввода/вывода. Это одна из особенностей работы по сети в Java.

IP-адреса недостаточно для уникальной идентификации сервера, т. к. на одной машине может существовать несколько серверов. Каждая IP-машина также содержит порты, и когда устанавливается клиент или сервер, тогда необходимо выбрать порт, через который и клиент, и сервер согласны соединиться.

Порт – это программная абстракция. Клиентская программа знает, как соединится с машиной через ее IP-адрес, но она не может соединиться с определенной службой на этой машине. Таким образом, номер порта стал вторым уровнем адресации. Идея состоит в том, что при запросе определенного порта запрашивается служба, ассоциированная с этим номером порта. Обычно каждая служба ассоциируется с уникальным номером порта на определенной серверной машине. Клиент должен предварительно знать, на каком порту запущена нужная ему служба.

Системные службы зарезервировали использование портов с номерам от 1 до 1024, так что вы не можете использовать этот или любой другой порт, про который вы знаете, что он задействован.

Сокет – это программный интерфейс, предназначенный для передачи данных между приложениями. Что же касается типов сокетов, то их два – потоковые и датаграммные.

В Java создается сокет, чтобы создать соединение с другой машиной, затем получается InputStream и OutputStream (или, с соответствующими конверторами, Reader и Writer) из сокета, чтобы получить возможность трактовать соединение как объект потока ввода/вывода. Существует два класса сокетов, основанных на потоках: ServerSocket, который использует сервер для «прослушива-

ния» входящих соединений, и Socket, который использует клиент для инициализации соединения. Как только клиент создаст сокетное соединение, ServerSocket возвратит (посредством метода accept( )) соответствующий Socket, через который может происходить коммуникация на стороне сервера. На этой стадии используются методы getInputStream( ) и getOutputStream( ) для получения соответствующих объектов InputStream'a и outputstream'a для каждого сокета. Они должны быть обернуты внутрь буферных и форматирующих классов точно так же, как и другие объекты потоков.

Когда создается ServerSocket, то ему дается только номер порта и не дается ему IP-адрес, поскольку он уже есть на той машине, на которой он установлен. Однако когда создается Socket, то необходимо передать ему и IP-адрес, и номер порта, с которым необходимо соединиться. Socket, который возвращается из метода ServerSocket.accept( ), уже содержит всю эту информацию.

Представленный ниже пример покажет простейшее использование серверного и клиентского сокета. Все, что делает сервер, это ожидает соединения, затем использует сокет, полученный при соединении, для создания InputStream'a и OutputStream'a. Они конвертируются в Reader и Writer, которые оборачиваются в BufferedReader и PrintWriter. После этого все, что будет прочитано из BufferedReader'a, будет переправлено в PrintWriter, пока не будет получена строка "END", означающая, что пришло время закрыть соединение.

Клиент создает соединение с сервером, затем – OutputStream и некоторую обертку, как и в сервере. Строки текста посылаются через полученный PrintWriter. Клиент также создает InputStream с соответствующей конвертацией и оберткой, чтобы «слушать», что «говорит» сервер, который, в данном случае, просто отсылает слова назад.

Пример сервера, который просто отсылает назад все, что посылает клиент:

```
Import java.io.*;
import java.net.*;
public class ExampleServer {
  // Выбираем порт вне пределов 1–1024:
  public static final int PORT = 8080;
  public static void main(String[] args) throws
IOException {
  ServerSocket s = new ServerSocket(PORT);
  System.out.println("Started: " + s);
  try { /* Блокирует до тех пор, пока не возникнет соединение:*/
        Socket socket = s.accept();
        try {
System.out.println("Connection accepted:
" + socket);
BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getlnputStream())); /* Вывод
автоматически выталкивается из буфера PrintWriter'ом*/
        PrintWriter out = new PrintWriterlBufferedWriter(new
OutputStreamWriter(socket.getOutputStream())), true);
              while (true) {
String str = in.readLine();
```

```
        if (str.equals("END"))
        break;
System, out.println ("Echoing: " + str);
out.println(str);
}
/* Всегда закрываем два сокета...*/
   }
        finally {
.println(
System.out.println("closing...");
socket.close();
}
finally {
s. close ();
} } }
```

Таким образом, для ServerSocket'а необходим только номер порта, а не IP-адрес (т. к. он запускается на локальной машине!). Когда вызывается accept( ), метод блокирует выполнение до тех пор, пока клиент не попробует подсоединится к серверу. То есть сервер ожидает соединения, но другой процесс может выполняться. Когда соединение установлено, метод accept( ) возвращает объект Socket, представляющий это соединение.

В описанном механизме тщательно обработана ответственность за очистку сокета. Если конструктор ServerSocket завершится неудачей, программа просто завершится. По этой причине main( ) выбрасывает IOException, так что в блоке try нет необходимости. Если конструктор ServerSocket завершится успешно, то все вызовы методов должны быть помещены в блок «try-finally», чтобы убедиться, что блок не будет покинут ни при каких условиях и ServerSocket будет правильно закрыт.

Аналогичная логика используется для сокета, возвращаемого из метода accept( ). Если метод accept( ) завершится неудачей, то необходимо предположить, что сокет не существует и не удерживает никаких ресурсов, так что он не нуждается в очистке. Однако если он закончится успешно, то следующие выражения должны быть помещены в блок «try-finally», чтобы при каких-либо ошибках все равно произошла очистка. Позаботится об этом необходимо, потому что сокеты используют важные ресурсы, не относящиеся к памяти.

И ServerSocket, и Socket, производимый методом accept( ), печатаются в System.out. Это означает, что автоматически вызывается их метод toString( ). Вот, что выдает этот метод:

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

Следующая часть программы выглядит как открытие файла для чтения и записи, за исключением того, что InputStream и OutputStream создаются из объекта Socket. И объект InputStream'a, и OutputStream'a конвертируются в объекты Reader'a и Writer'a с помощью «классов-конвертеров» InputStreamReader и OutputStreamreader соответственно. Можно также использовать классы InputStream

и OutoutStream напрямую, но, с точки зрения вывода, есть явное преимущество в использовании этого подхода. Оно проявляется в PrintWriter'e, который имеет перегруженный конструктор, принимающий в качестве второго аргумента флаг типа boolean, указывающий, нужно ли автоматическое выталкивание буфера вывода в конце каждого выражения println( ) (но не print( )). Каждый раз при записи в вывод буфер вывода должен выталкиваться, чтобы информация проходила по сети. Выталкивание важно для этого конкретного примера, поскольку клиент и сервер ожидают строку от другой стороны, прежде чем приступят к ее обработке. Если выталкивание буфера не произойдет, информация не будет помещена в сеть до тех пор, пока буфер не заполнится, что может привести к многочисленным проблемам в представленном примере.

Когда пишется сетевая программа, необходимо быть осторожным при использовании автоматического выталкивания буфера. При каждом выталкивании буфера пакеты должны создаваться и отправляться. Другими словами, конец строки является концом сообщения. Но во многих случаях сообщения не ограничиваются строками, так что будет более эффективным использовать автоматическое выталкивание буфера. Для этого используется встроенный механизм буфферизации для построения и отсылки пакета. В таком случае могут быть посланы пакеты большего размера и процесс обработки пойдет быстрее.

Все открытые в примере потоки являются буфферезированными.

В бесконечном цикле while происходит чтение строк из входного BufferedReader'a и запись информации в System.out и в выходной PrintWriter.

Вход и выход могут быть любыми потоками, связанными с сетью.

Когда клиент посылает строку, содержащую «END», программа прекращает цикл и закрывает сокет.

Пример клиента, который просто посылает строки на сервер и читает строки, посылаемые сервером:

```
import java.net.*;
import java.io.*;

public class ExampleClient {
public static void main(String[] args) throws IOException {
// Передаем null в getByName(), получая
// специальный IP адрес "локальной заглушки"
// для тестирования на машине без сети:
InetAddress addr = InetAddress.getByName(null);
// Альтернативно, вы можете использовать
// адрес или имя:
// InetAddress addr =
// InetAddress.getByName("127.0.0.1");
// InetAddress addr =
// InetAddress.getByName("localhost")
System.out.println("addr = " + addr);
Socket socket = new Socket(addr, JabberServer.PORT);
// Помещаем все в блок try-finally, чтобы
// быть уверенным, что сокет закроется:
try {
```

```
        System.out.println("socket = " + socket)
        BufferedReader in = new BufferedReader(new InputStreamReader(socket
    .getInputStream()));
    // Вывод автоматически выталкивается PrintWriter'ом.
    PrintWriter out = new PrintWriter(new BufferedWriter(
                        new OutputStreamWriter(socket.getOutputStream())), true);
       for (int i = 0; i < 10; i++) {
       out.println("howdy " + i) ;
       String str=in.readLine();
       System.out.println(str);
       }
       out.println("END");
    }
       finally {
            System.out.println("closing...");
            socket.close();
          }
       }
    }
```

В main( ) представлены все три способа получения IP-адреса локальной заглушки: с помощью null, localhost или путем явного указания зарезервированного адреса 127.0.0.1. Чтобы соединится с машиной по сети, необходимо заменить зарезервированный адрес IP-адресом машины.

Socket создается при указании и InetAddress'a, и номера порта. Интернет-соединение уникально определяется четырьмя параметрами: клиентским хостом, клиентским номером порта, серверным хостом и серверным номером порта. Когда запускается сервер, он получает назначаемый порт (8080) на localhost (127.0.0.1). Когда запускается клиент, он располагается на следующем доступном порту на своей машине, 1077 – в данном случае порт, который оказался на той же машине (127.0.0.1), что и сервер. Теперь, чтобы передать данные между клиентом и сервером, каждая сторона знает, куда посылать их. Поэтому в процессе соединения с «известным» сервером клиент посылает «обратный адрес», чтобы сервер знал, куда посылать данные. Вот что представлено среди выводимого стороной сервера:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

Это означает, что сервер принимает соединение с адреса 127.0.0.1 и порта 1077 во время прослушивания локального порта (8080). На клиентской стороне:

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

Это значит, что клиент установил соединение с адресом 127.0.0.1 по порту 8080, используя локальный порт 1077.

При каждом повторном запуске клиента номер локального порта увеличивается. Он начинается с 1025 (первый после зарезервированного блока портов)

и будет увеличиваться до тех пор, пока пользователь не перезапустит машину, в таком случае он снова начнется с 1025. На машинах под управлением UNIX, как только будет достигнут верхний предел диапазона сокетов, номер будет возвращен снова к наименьшему доступному номеру.

Как только объект Socket будет создан, процесс перейдет к BufferedReader и PrintWriter, как было представлено в сервере (в обоих случаях создание соединения начинаете с Socket'a). В данном случае клиент инициирует обмен путем посылки строки "howdy", за которой следует число. Буфер должен опять выталкиваться (это происходит автоматически из-за второго аргумента в конструкторе PrintWriter'a). Если буфер не будет выталкиваться, процесс обмена повиснет, поскольку начальное "howdy" никогда не будет послано (буфер недостаточно заполнен, чтобы отсылка произошла автоматически). Каждая строка, посылаемая назад сервером, записывается в System.out, чтобы проверить, что все работает корректно. Для завершения обмена посылается ранее оговоренный «END». Если клиент просто разорвет соединение, то сервер выбросит исключение.

Аналогичные меры приняты, чтобы сетевые ресурсы, представляемые сокетом, правильно очищены. Для этого используется блок «try-finally».

**Задание**

Создать на основе сокетов клиент-серверное приложение.

1 Клиент посылает через сервер сообщение другому клиенту.

2 Клиент посылает через сервер сообщение другому клиенту, выбранному из списка.

3 Чат. Клиент посылает через сервер сообщение, которое получают все клиенты. Список клиентов хранится на сервере в файле.

4 Клиент при обращении к серверу получает случайно выбранный сонет Шекспира из файла.

5 Сервер рассылает сообщения выбранным из списка клиентам. Список хранится в файле.

6 Сервер рассылает сообщения в определенное время определенным клиентам.

7 Сервер рассылает сообщения только тем клиентам, которые в настоящий момент находятся в on-line.

8 Чат. Сервер рассылает всем клиентам информацию о клиентах, вошедших в чат и покинувших его.

9 Клиент выбирает изображение из списка и пересылает его другому клиенту через сервер.

10 Игра по сети в «Морской бой».

11 Игра по сети «Го». Крестики-нолики на безразмерном (большом) поле. Для п1обеды необходимо выстроить пять в один ряд.

12 Написать программу, сканирующую сеть в указанном диапазоне IP-адресов на наличие активных компьютеров.

13 Прокси. Программа должна принимать сообщения от любого клиента, работающего на протоколе TCP, и отсылать их на соответствующий сервер. При передаче изменять сообщение.

14 Телнет. Создать программу, которая соединяется с указанным сервером по указанному порту и производит обмен текстовой информацией.

Вариант назначается преподавателем.

***Контрольные вопросы***

1 Что такое архитектура клиент-сервер?
2 Назначение порта, зарезервированные номера портов.
3 Что такое сокет, назначение сокета?
4 Опишите механизм создания входного и выходного потоков.
5 Опишите механизм работы метода accept().

# 4 Лабораторная работа № 4. Сервлеты

**Цель работы**: создать сервлет и взаимодействующие с ним пакеты Java-классов и JSP-страницы.

**Порядок выполнения работы.**

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.
2 Получить задание у преподавателя, выполнить задание.
3 Оформить отчет.

**Требования к отчету.**

1 Цель работы.
2 Результаты работы.
3 Выводы.

***Теоретические сведения***

Клиентский доступ из интернета или корпоративного интранета является легким способом получить или предоставить доступ к данным и ресурсам. Этот тип доступа основывается на клиентах, использующих стандартный для World Wide Web Язык Гипертекстовой Разметки (Hypertext Markup Language – HTML) и Протокол Передачи Гипертекста (Hypertext Transfer Protocol – HTTP). Сервлетное API является набором абстрактных общих решений рабочей среды, соответствующей HTTP-запросам.

Традиционно, способ решения таких проблем, как возможность для Интернет-клиента обновления базы данных, состояла в создании HTML-страницы с текстовыми полями и кнопкой «Submit».

Пользователь впечатывал соответствующую информацию в текстовые поля и нажимал кнопку «Submit». Данные передавались вместе с URL, который говорил серверу, что делать с данными, указывая расположение программы Common Gateway Interface (CGI), которую запускал сервер, обеспечивая про-

грамму данными, которые она требовала. CGI-программа обычно написана на Perl, Python, C, C++ или любом другом языке, который умеет читать стандартный ввод и писать в стандартный вывод. Стандартное предоставление Web-сервером: вызывалась CGI-программа и использовались стандартные потоки (или, как возможный вариант для ввода, использовалась переменная окружения) для ввода и вывода. CGI-программа отвечала за все остальное. Во-первых, она просматривала данные и решала, является ли их формат корректным. Если нет, CGI-программа должна создать HTML-страницу для описания проблемы; эта страница передавалась Web-серверу (через стандартный вывод из CGI программы), который посылал ее назад пользователю. Пользователь должен был вернуться на страницу и попробовать снова. Если данные были корректными, CGI-программа обрабатывала данные соответствующим способом, возможно, добавляла их в базу данных. Затем она должна была произвести соответствующую HTML-страницу для Web-сервера, чтобы он вернул ее пользователю.

С развитием технологий возникла необходимость в переходе к решению, полностью основанному на Java: апплет на клиентской стороне для проверки и пересылки данных и сервлет на серверной стороне для приема и обработки данных. Апплеты предоставляют технологию с вполне достаточной поддержкой, но являются проблематичными в использовании на Web, поскольку нет возможности рассчитывать на поддержку определенной версии Java на клиентском Web-браузере, т. е. невозможно рассчитывать, что Web-браузер поддерживает Java. В Интранете можно потребовать, чтобы определенная поддержка Java была доступна, что позволит реализовать намного большую гибкость в том, что можно сделать, но в Web более безопасный подход состоит в том, чтобы вся обработка была на стороне сервера, а клиенту доставлялся обычный HTML.

Таким образом, ни один пользователь не будет отклонен сайтом по причине того, что у него нет правильно установленного программного обеспечения.

Поскольку сервлеты предоставляют решение для программной поддержки на стороне сервера, они являются одной из наиболее популярных причин перехода на Java. Не только потому, что они предоставляют рабочую среду, которая заменяет CGI-программирование, но весь ваш код приобретает портируемость между платформами, получаемую от использования Java, и пользователь приобретает доступ ко всему Java API (за исключением, конечно, того, которое производит GUI, такого языка, как Swing).

Архитектура API-сервлета основывается на том, что классический провайдер сервиса использует метод service( ), через который все клиентские запросы будут посылаться программным обеспечением контейнера сервлетов, и методы жизненного цикла init( ) и destroy( ), которые вызываются только в то время, когда сервлет загружается и выгружается (это случается редко).

```
public interface Servlet {
public void init(ServletConfig config) throws ServletException;
```

```
public ServletConfig getServletConfig();

public void service(ServletRequest req, ServletResponse res);
            throws ServletException, IOException;
public String getServletInfo();
public void destroy();
}
```

Основное назначение getServletConfig( ) состоит в возвращении объекта ServletConfig, который содержит параметры инициализации и запуска для этого сервлета, getServletInfo( ) возвращает строку, содержащую информацию о сервлете, такую, как автор, версия и авторские права.

Класс GenericServlet является оболочечной реализацией этого интерфейса и обычно не используется. Класс HttpServlet является расширением GenericServlet и специально предназначен для обработки HTTP-протокола – HttpServelt является одним из тех классов, которые используется чаще всего.

Основное удобство атрибутов сервлетного API состоит во внешних объектах, которые вводятся вместе с классом HttpServlet для его поддержки. Если взглянуть на метод service( ) в интерфейсе Servlet, то можно увидеть, что он имеет два параметра: ServeltRequest и ServletResponse. Вместе с классом HttpServlet эти два объекта предназначены для HTTP: HttpServletRequest и HttpServletResponse.

Ниже придставлен простейший пример, который показывает использование HttpServletResponse:

```
// (Depends: j2ee.jar)
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
    public class ServletsRule extends HttpServlet {
      int i = 0; // "постоянство" сервлета
public void service(HttpServletRequest req,HttpServletResponse res)
                    throws IOException {
res.setContentType("text/html");
PrintWriter out = res.getWriter();
out. print ("<HEAD><TITLE>");
out.print("A server-side strategy");
out. print ("</TITLEX/HEADXBODY>");
out.print("<hl>Servlets Rule! " + i++);
out.print("</hl></BODY>");
out.close();
} }
```

Программа ServletsRule настолько проста, насколько может быть прост сервлет. Сервлет инициализируется только один раз путем вызова его метода init( ), при загрузке сервлета после того, как контейнер сервлетов будет загружен в первый раз. Когда клиент создает запрос к URL, который представлен сервлетом, контейнер сервлетов перехварывает этот запрос и совершает вызов

метода service( ) после установки объектов HttpServletRequest и HttpServletResponse.

Основная ответственность метода service( ) состоит во взаимодействии с HTTP-запросом, который посылает клиент, и в построении HTTP-ответа, основываясь на атрибутах, содержащихся в запросе. ServletsRule манипулирует только объектом ответа, не обращая внимания на то, что посылает клиент.

После установки типа содержимого клиента (которое должно всегда выполнятся прежде, чем будет получен Writer или OutputStream), метод getWriter( ) объекта ответа производит объект PrintWriter, который используется для записи символьных данных ответа (другой вариант: getOutputStream( ) производит OutputStream, используемый для бинарного ответа, который применим для более специализированных решений).

Оставшаяся часть программы просто посылает HTML клиенту (предполагается, что пользователь владеет HTML), как последовательность строк. Однако, необходимо обратить внимание на включение «счетчика показов», представленного переменной i. Он автоматически конвертируется в строку в инструкции print( ).

Когда пользователь запустит программу, то увидит, что значение i сохраняется между запросами к сервлету. Это важное свойство сервлетов: т. к. только один сервлет определенного класса загружается в контейнер, и он никогда не выгружается (до тех пор, пока контейнер не завершит свою работу, что обычно случается только при перезагрузке серверного компьютера), любые поля сервлета этого класса действительно становятся постоянными объектами. Это значит, что пользователь может без усилий сохранять значения между запросами к сервлету, в то время как в CGI пользователь должен записывать значения на диск, чтобы сохранить их, что требует большого количества окружения для правильного их получения, а в результате получается не кроссплатформенное решение.

Конечно, иногда Web-сервер, а таким образом и контейнер сервлетов, должны перегружаться. Для предотвращения потери любой постоянной информации методы сервлета init( ) и destroy( ) автоматически вызываются во время загрузки или выгрузки клиента, что дает пользователю возможность сохранить важные данные во время остановки и восстановления после перезагрузки. Контейнер сервлетов вызывает метод destroy( ), когда он завершает свою работу, так что пользователь всегда получает возможность сохранить значимые данные до тех пор, пока серверная машина не будет сконфигурирована разумным способом.

Есть другая особенность при использовании HttpServlet. Этот класс обеспечивает методы doGet( ) и doPost( ), которые приспособлены для CGI "GET" пересылки от клиента и CGI "POST". GET и POST отличаются только деталями в способе пересылки данных. Однако большинство доступной информации поддерживает создание раздельных методов doGet( ) и doPost( ) вместо единого общего метода service( ), который обрабатывает оба случая. В следующем примере используется метод service( ), который заботится о выборе GET или POST.

Когда бы форма не отсылалась сервлету, HttpServletRequest поступает со

всеми предварительно загруженными данными формы, хранящимися как пары «ключ – значение». Если известно имя поля, то можно просто использовать его напрямую в методе getParameter( ), чтобы найти значение. Можно также получить Enumiraton (старая форма Итератора) из имен полей, как показано в следующем примере. Этот пример также демонстрирует, как один сервлет может быть использован для воспроизведения страницы, содержащей форму, и для страницы ответа. Если Enumiration пустое, значит нет полей. Это означает, что форма не была отправлена. В данном случае производится форма, а кнопка подтверждения будет повторно вызывать тот же самый сервлет. Однако если поля существуют, они будут отображены, как представлено в примере:

```
// Группа пар имя–значение любой HTML формы
// {Depends: j2ee.jar}
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class EchoForm extends HttpServlet {
public void service(HttpServletRequest req,HttpServletResponse res)
                    throws IOException {
res.setContentType("text/html");
PrintWriter out = res.getWriter();
Enumeration fids = req.getParameterNames();
if (!fids.hasMoreElements()) {
       // Нет отосланной формы -- создаем:
out.print("<html>");
out.print("<form method="POST"" + "action="EchoForm">");
for (int i = 0; i < 10; i++)
         out.print("<b>Field" + i + "</b> "<input type="text""
+ " size="20" name="Field" + i + ""value="Value" + i + ""><br>");
         out.print("<INPUT TYPE=submit name=submit"
+ " Value="Submit"></form></html>");
}
    else {
          out.print("<hl>Your form contained:</hl>");
          while (flds.hasMoreElements()) {
                String field = (String) flds.nextElement();
                String value = req.getParameter(field);
                out.print(field + " = " + value +"<br>");
          }
       }
       out.close();
   } }
```

В Java не удобно обрабатывать строки в памяти – форматирование возвращаемой страницы достаточно тягостно из-за символов завершения строки, эскейп-символов и знаков «+», необходимых для построения объектов String. Для огромных HTML-страниц становится неразумным помещение кода прямо в Java. Одно из решений состоит в хранении страницы, как отдельного текстового файла, который потом открывается и передается Web-серверу. Если выпол-

нить замену любого вида для содержимого этой страницы, то Java плохо обработает строки. В этом случае, вероятно, лучше использовать более подходящее решение (Python может быть таким решением; существует версия, которая сама встраивается в Java и называется JPython) для генерации страницы ответа.

**Задание**

Создать сервлет и взаимодействующие с ним пакеты Java-классов и JSP-страницы, выполняющие следующие действия.

1 Генерация таблиц по переданным параметрам: заголовок, количество строк и столбцов, цвет фона.

2 Вычисление тригонометрических функций в градусах, радианах с указанной точностью. Выбор функций должен осуществляться через выпадающий список.

3 Поиск слова, введенного пользователем. Поиск и определение частоты встречаемости осуществляется в текстовом файле, расположенном на сервере.

4 Вычисление объемов тел (параллелепипед, куб, сфера, тетраэдр, тор, шар, эллипсоид и т. д.) с точностью и параметрами, указываемыми пользователем.

5 Поиск и (или) замена информации в коллекции по ключу (значению).

6 Выбор текстового файла из архива файлов по разделам (поэзия, проза, фантастика и т. д.) и его отображение.

7 Выбор изображения по тематике (природа, автомобили, дети и т. д.) и его отображение.

8 Информация о среднесуточной температуре воздуха за месяц задана в виде списка, хранящегося в файле. Определить: среднемесячную температуру воздуха; количество дней, когда температура была выше среднемесячной; количество дней, когда температура опускалась ниже 0 °С; три самых теплых дня.

9 Определение значения полинома в заданной точке. Степень полинома и его коэффициенты вводятся пользователем.

10 Вывод фрагментов текстов шрифтами различного размера. Размер шрифта и количество строк задаются на стороне клиента.

11 Информация о точках на плоскости хранится в файле. Выбрать все точки, наиболее приближенные к заданной прямой. Параметры прямой и максимальное расстояние от точки до прямой вводятся на стороне клиента.

12 Осуществить сортировку введенного пользователем массива целых чисел. Числа вводятся через запятую.

13 Реализовать игру с сервером в крестики-нолики.

Вариант назначается преподавателем.

***Контрольные вопросы***

1 Дайте пояснение сервлетному API.

2 Назначение программы CGI.

3 В чем заключается недостаток апплетов?

4 В чем заключается достоинство сервлетов?

5 Перечислите основные методы жизненного цикла сервлетов.

# 5 Лабораторная работа № 5. Соединение с базой данных

**Цель работы**: приобрести навыки проектирования и разработки приложений в архитектуре клиент-сервер.

**Порядок выполнения работы.**

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.

3 Оформить отчет.

**Требования к отчету.**

1 Цель работы.

2 Результаты работы.

3 Выводы.

***Теоретические сведения***

JDBC (Java DataBase Connectivity) – стандартный прикладной интерфейс (API) языка Java для организации взаимодействия между приложением и СУБД. Это взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов.

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным базам данных (БД), для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Строго говоря, JDBC не имеет прямого отношения к J2EE, но так как взаимодействие с СУБД является неотъемлемой частью Web-приложений, то эта технология рассматривается в данном контексте.

Последовательность действий при работе с JDBC.

1 Загрузка класса драйвера базы данных при отсутствии экземпляра этого класса. Например:

```
String driverName = "org.gjt.mm.mysql.Driver;
для СУБД MySQL,
String driverName = "sun.jdbc.odbc.JdbcOdbcDriver";
для СУБД MSAccess или
String driverName = "org.postgreesql.Driver";
для СУБД PostgreeSQL.
```

После этого выполняется собственно загрузка драйвера в память:

```
Class.forName(driverName);
```

Тогда становится возможным соединение с СУБД.

Эти же действия можно выполнить, импортируя библиотеку и создавая объект явно. Например, для СУБД DB2 от IBM объект-драйвер можно создать

следующим образом:

```
new com.ibm.db2.j dbc.net.DB2Driver();
```

2 Установка соединения с БД.

Для установки соединения с БД вызывается статический метод getConnection() класса DriverManager. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Метод возвращает объект Connection. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов. Например:

```
Connection cn = DriverManager.getConnection(
"jdbc:mysql://localhost/my_db", "root", "pass");
```

В результате будет возвращен объект Connection и будет одно установленное соединение с БД my db. Класс DriverManager предоставляет средства для управления набором драйверов баз данных. С помощью метода registerDriverO драйверы регистрируются, а методом getDriversQ можно получить список всех драйверов.

3 Создание объекта для передачи запросов.

После создания объекта Connection и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект Statement, создаваемый вызовом метода createStatement() класса Connection.

```
Statement st = cn.createStatement();
```

Объект класса Statement используется для выполнения SQL-запроса без его предварительной подготовки. Могут применяться также объекты классов PreparedStatement и CallableStatement для выполнения подготовленных запросов и хранимых процедур. Созданные объекты можно использовать для выполнения запроса SQL, передавая его в один из методов executeQuery(String sql) или executeUpdate(String sql).

4 Выполнение запроса.

Результаты выполнения запроса помещаются в объект ResultSet:

```
ResultSet rs = st.executeQuery("SELECT * FROM my_table");
//выборка всех данных таблицы my_table
```

Для добавления, удаления или изменения информации в таблице вместо метода executeQuery() запрос помещается в метод executeUpdate().

5 Обработка результатов выполнения запроса.

Обработка результатов производится методами интерфейса ResultSet, где самыми распространенными являются next() и getString(int pos) а также аналогичные методы, начинающиеся с getТип(int pos) (getlnt(int pos), getFloat(int pos) и др.) и updateТип ().Среди них следует выделить методы getClob(int pos) и get-

Blob(intpos), позволяющие извлекать из полей таблицы специфические объекты (Character Large Object, Binary Large Object), которые могут быть, например, графическими или архивными файлами. Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его позиции в строке.

При первом вызове метода next() указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение false.

6 Закрытие соединения.

```
cn.close ();
```

После того как база больше не нужна, соединение закрывается.

Для того чтобы правильно пользоваться приведенными методами. программисту требуется знать типы полей БД. В распределенных системах это знание предполагается изначально.

Теперь следует воспользоваться всеми предыдущими инструкциями и создать пользовательскую БД с именем db2 и одной таблицей users. Таблица должна содержать два поля: символьное – name и числовое – phone и несколько занесенных записей. Сервлет, осуществляющий простейший запрос на выбор всей информации из таблицы, выглядит следующим образом:

```
import java.io.*;
import java.sql.*;
import javax.servlet
import javax.servlet.http.*;

public class ServletToBase extends HttpServlet {
public void doGet(HttpServletRequest req,
HttpServletResponse resp)
                        throws ServletException {
performTask(req, resp);
}
public void doPost(HttpServletRequest req,
HttpServletResponse resp)
                        throws ServletException {
performTask(req, resp);
}
public void showInfo(PrintWriter out, ResultSet rs)
                        throws SQLException {
      out.print("From DataBase:")
while (rs.next()) {
      out.print("<br>Name:–> " + rs.getString(1) + " Phone:–> " + rs.getInt(2));
                    }
}
public void performTask(HttpServletRequest req,
HttpServletResponse resp) {
resp.setContentType("text/html; charset=Cpl251");
PrintWriter out = null;
```

```
        try {//1
            out = resp.getWriter();
            try {//2

    Class . forName ("org.gjt .mm.mysql .Driver") ;
    // для MSAccess
            /* return "sun.jdbc.odbc.JdbcOdbcDriver' */
    // для PostgreeSQL
            /* return " org.postgreesql.Driver " */
    Connection cn = null;
            try {//3
    cn =
    DriverManager .getConnection("jdbc:mysql://localhost/db2","root", "pass");
    // для MSAccess
            /* return "jdbc:odbc:db2"; */
    // для PostgreeSQL
            /* return "jdbc:postgreesql://localhost/db2"; */

    Statement st = null;
            try {//4
    st = cn. createStatement ();
    ResultSet rs = null;
            try {//5
    rs = st.executeQuery("SELECT * FROM users");
          out.print("From DataBase:");
    while (rs.next()) {
          out.print("<br>Name:–> " + rs.getString(1) + " Phone:-> " + rs.getInt(2));
                }
    } finally { // для 5-го блока try
      /*    закрыть ResultSet, если он был открыт и ошибка произошла во время
чтения из него данных */
                    // проверка успел ли создаться ResultSet
    if (rs != null) rs.close();
    else
    out.print("ошибка во время чтения данных из БД");
             }
               } finally { // для 4-го блока try
      /*    закрыть Statement, если он был открыт и ошибка произошла во время
создания ResultSet */
                    // проверка успел ли создаться Statement
    if (st != null) st.close();
    else
    out.print("Statement не создан");
             }
             } finally {// для 3-го блока try
    /* закрыть Connection, если он был открыт и ошибка произошла во время созда-
ния ResultSet или создания и использования Statement */
                   // проверка успел ли создаться Connection
    if (cn != null) cn.close();
    else
    out.print("Connection не создан");
            }
```

```
            } catch (ClassNotFoundException e) {// для 2-гоблока try
out.print("ошибка во время загрузки драйвера БД");
                                                    }
                        }
/* вывод сообщения о всех SQLException и IOException в блоках finally, */
      /* поэтому следующие блоки catch оставлены пустыми */
            catch (SQLException e) {
            }// для 1-го блока try
            catch (IOException e) {
            }// для 1-го блока try
            finally {// для 1-го блока try
/* закрыть PrintWriter, если он был инициализирован и ошибка произошла во
время работы с БД */
// проверка, успел ли инициализироваться PrintWriter
if (out != null) out.close();
else
out.print("PrintWriter не проинициализирован");
                        }
      }
}
```

В несложном приложении достаточно контролировать закрытие соединения, т. к. незакрытое («провисшее») соединение снижает быстродействие системы.

**Задание**

В соответствии с заданием, выданным преподавателем, необходимо выполнить следующие действия.

1 Организацию соединения с базой данных вынести в отдельный класс, метод которого возвращает соединение.

2 Создать БД. Привести таблицы к одной из нормированных форм.

3 Создать класс для выполнения запросов на извлечение информации из БД с использованием компилированных запросов.

4 Создать класс на добавление информации.

5 Создать HTML-документ с полями для формирования запроса.

6 Результаты выполнения запроса передать клиенту в виде HTML-документа.

***Контрольные вопросы***

1 Дайте определение JDBC.

2 Опишите последовательность действий при работе с JDBC.

3 Поясните работу сервлета, осуществляющего простейший запрос на выбор всей информации из созданной таблицы.

# 6 Лабораторная работа № 6. Удаленный вызов метода (RMI)

**Цель работы**: реализовать приложение, используя технологию RMI.

**Порядок выполнения работы.**

1 Изучить основные теоретические положения, сделав необходимые выписки в конспект.

2 Получить задание у преподавателя, выполнить задание.

3 Оформить отчет.

**Требования к отчету.**

1 Цель работы.

2 Результаты работы.

3 Выводы.

***Теоретические сведения***

Удаленный вызов метода (RMI) дает возможность выполнять объекты Java на различных компьютерах или в отдельных процессах путем взаимодействия их друг с другом посредством удаленных вызовов методов. Технология RMI основана на более ранней подобной технологии удаленного вызова процедур (RPC) для процедурного программирования, разработанной в 80-х гг. RPC позволяет процедуре вызывать функцию на другом компьютере столь же легко, как если бы эта функция была частью программы, выполняющейся на том же компьютере. RPC выполняет всю работу по организации сетевых взаимодействий и маршалинга данных (т. е. пакетирования параметров функций и возврата значений для передачи их через сеть). Но RPC не подходит для передачи и возврата объектов Java, потому что она поддерживает ограниченный набор простых типов данных. Есть и другой недостаток у RPC-программисту необходимо знать специальный язык определения интерфейса (IDL) для описания функций, которые допускают удаленный вызов. Для устранения этих недостатков и была разработана технология RMI.

RMI делает тяжелым использование интерфейсов. Когда пользователь желает создать удаленный объект, он маскирует лежащую в основе реализацию, общаясь по интерфейсу. То есть, когда клиент получает ссылку на удаленный объект, реально он получает ссылку на интерфейс, который соединен с некоторым локальным кодом заглушки, которая общается по сети. Но пользователь не задумывается об этом, он просто посылает сообщения через ссылку на интерфейс.

Когда пользователь создает удаленный интерфейс, он следует следующему руководству.

1 Удаленный интерфейс должен быть публичным (он не может иметь «пакетный уровень доступа», таким образом, он не может быть «дружественным»). В противном случае, клиент получит ошибку при попытке загрузить удаленный объект, который реализует удаленный интерфейс.

2 Удаленный интерфейс должен наследоваться от интерфейса java.rmi.Remote.

3 Каждый метод удаленного интерфейса должен декларировать java.rmi.RemoteException в своем разделе throws в дополнение ко всем остальным специфичным для приложения исключениям.

4 Удаленный объект, передающийся как аргумент или возвращаемое значение (либо напрямую, либо как встроенная часть локального объекта), должен быть декларирован как удаленный интерфейс, а не класс реализации.

Вот простейший удаленный интерфейс, который представляет службу точного времени:

```
// The PerfectTime remote interface
package c15.rmi;
import java.rmi.*;

public interface PerfectTimel extends Remote {
long getPerfectTime() throws RemoteException;
}
```

Интерфейс в примере выглядит как и любой другой интерфейс, за исключением того, что он является наследником Remote и все его методы выбрасывают RemoteException. Все методы интерфейса автоматически являются публичными.

Сервер должен содержать класс, который наследуется от UnicastRemoteObject и реализует удаленный интерфейс. Этот класс может также иметь дополнительные методы, но для клиента, конечно, будут доступны только методы удаленного интерфейса, так как клиент будет получать только ссылку на интерфейс, а не на класс, реализующий его.

Необходимо явно определить конструктор для удаленного объекта, даже если определвен только конструктор по умолчанию, который вызывает конструктор базового класса. Необходимо написать его, так как он должен выбрасывать RemoteException.

Реализация удаленного интерфейса PerfectTimel представлена в следующем примере:

```
// Реализация удаленного
// объекта PerfectTime.
// {Broken}
package c15.rmi;
import java.rmi.*;
import ]java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;
public class PerfectTime extends UnicastRemoteObject
implements PerfectTimel {
   // Реализация интерфейса:
   public long getPerfectTime() throws RemoteException {
   return System.currentTimeMillis();
```

```
}
   // Необходимо реализовывать конструктор,
   // выбрасывающий RemoteException:
   public PerfectTime() throws RemoteException {
   super (); // Вызывается автоматически
}
// Регистрация для обслуживания RMI. Выбрасывает
// исключения на консоль.
   public static void main(String[] args) throws Exception {
         System.setSecurityManager(new RMISecurityManager());
         PerfectTime pt = new PerfectTime();
         Naming.bind("//peppy:2005/PerfectTime", pt)
         System.out.println("Ready to do time");
      }
}
```

Здесь main( ) обрабатывает все детали установки сервера. При обслуживании RMI объектов, в некотором месте программы необходимо выполнить следующее.

1 Создать и установить менеджер безопасности, который поддерживает RMI. Для RMI досупен только один менеджер, как часть пакета Java, он называется RMISecurityManager.

2 Создать один или несколько экземпляров удаленного объекта. В приведенном примере создан объект PerfectTime.

3 Зарегистрировать как минимум один удаленный объект в RMI репозитории удаленных объектов в целях загрузки. Один удаленный объект может иметь методы, которые производят ссылки на другие удаленные объекты. Это позволит пользователю сделать так, чтобы клиент обращался к репозиторию только один раз, чтобы получить первый удаленный объект.

В этом примере осуществлен вызов статического метода Naming.bind( ). Однако этот вызов требует, чтобы репозиторий был запущен как отдельный процесс на компьютере. Имя сервера репозитория rmiregistry в системе Windows используется команда start rmiregistry для запуска его в фоновом режиме. Для системы Unix используется команда rmiregistry &.

Как и многие сетевые программы, rmiregestry получает IP адрес той машины, на которой она запущена, но она также должна «слушать» порт. Если вызвать rmiregestry как это показано выше, без аргументов, для репозитория будет использован порт по умолчанию 1099. Если необходимо, чтобы репозиторий был на другом порту, то нужно добавить аргумент в командную строку, который указывает порт. Например, интересующий порт 2005, так что rmiregestry должна быть запущена следующей командой под Windows: start rmiregistry 2005, или для Unix: rmiregistry 2005 &.

Информация о номере порта также должна быть передана в команду bind( ), точно так же, как и IP-адрес машины, где расположен репозиторий.

Существуют некоторые особенности работы с RMI.

1 localhost не работает с RMI. То есть, экспериментируя с RMI на одной единственной машине, необходимо предоставить имя машины. Чтобы найти

имя используемой машины в среде Windows, нужно запустить контрольную панель и выберать «Network». При выборе закладки «Identification» можно увидеть имя используемого компьютера.

2 RMI не будет работать до тех пор, пока используемый компьютер не будет иметь активного TCP/IP-соединения, даже если все компоненты просто общаются друг с другом на используемой локальной машине. Это означает, что необходимо соединить компьютер с соответствующим провайдером Internet, прежде чем пробовать запускать программу или будет получено непонятное сообщение об исключении.

Учитывая все вышесказанное, команда bind( ) принимает вид

```
Naming.bind("//peppy:2005/PerfectTime", pt);
```

Если используется порт по умолчанию 1099, то не нужно указывать порт, поэтому можно написать следующее:

```
Naming.bind("//peppy/PerfectTime", pt);
```

Пользователь должен быть способен выполнить локальное тестирование, оставив в покое IP-адрес и используя только идентификатор:

```
Naming .bind ("PerfectTime", pt);
```

Имя службы произвольно, в данном случае это PerfectTime, что совпадает с именем класса, но также возможно использовать любое название. Важно то, что оно должно быть уникальным в известном клиенту репозитории, к которому он обращается для производства удаленного объекта. Если имя уже есть в репозитории, то ответ будет следующий: AlreadyBoundException. Чтобы предотвратить это, можно всегда использовать rebind( ) вместе bind( ), т. к. rebind( ) либо добавляет новую запись, либо заменяет уже существующую.

Даже после того, как main( ) завершит работу, объект будет создан и зарегистрирован, он останется «жить» в репозитории, ожидая прихода клиентов и запросов от них. До тех пор, пока работает nniregistry и не вызван метод Naming.unbind( ) с присвоенным именем, объект будет существовать. По этой причине, когда разрабатывается код, необходимо выгружать rmiregistry и перезапускать его, когда скомпилирована новая версия удаленного объекта.

Нет необходимости принудительно запускать rmiregistry, как внешний процесс. Если известно, что разрабатываемое приложение является единственным, использующим репозиторий, то можно запускать его внутри программы с помощью следующей строки:

```
LocateRegistry.createRegistry(2005);
```

Как и прежде, 2005 – это номер порта, который использовался в примере. Это эквивалентно запуску rmiregistry2005 из командной строки, но такой запуск часто будет более последовательным, при разрабатке RMI кода, т. к. при этом

пропускаются дополнительные шаги запуска и остановки репозитория. Как только выполнится этот код, можно вызвать bind( ), используя Naming.

Если скомпилировать и запустить PerfectTime.java, программа не будет работать, даже если репозиторий запущен правильно. Это происходит потому, что рабочая среда для RMI еще не до конца создана. Необходимо сначала создать заглушки и скелеты, которые обеспечат операцию сетевого соединения и позволят считать, что удаленный объект – это просто другой локальный объект на рабочей машине.

Любые объекты, которые передаются при вызове или получении от удаленного объекта должны реализовывать Serializable (если необходимо передавать удаленную ссылку вместо самого объекта, объектные аргументы могут реализовывать Remote), так что можно представить, что заглушки и скелеты автоматически выполняют сериализацию и десериализацию, как будто они «руководят» всеми аргументами через сеть и возвращают результат. Поэтому необходимо создать заглушки и скелеты. Это простой процесс: нужно вызвать инструмент rmic для скомпилированного кода и создать необходимые файлы. Таким образом, требуется просто добавить еще один шаг в процесс компиляции.

Однако инструмент rmic на распознает пакеты и classpaths. PerfectTime.java расположен в пакете c15.rmi, и если вызвать rmic в том же директории, в котором расположен PerfectTime.class, rmic не найдет файл, так как он ищет в переменной classpath. Таким образом, нужно указать расположение в переменной classpath, например: rmic c15.rmi.PerfectTime.

Нет необходимости находиться в директории, содержащей PerfectTime.class, когда запускается эта команда, а результат будет помещен в текущий каталог.

Когда запуск rmic завершится успешно, будут получены два новых класс в директории:

```
PerfectTime_Stub.class
PerfectTime_Skel.class
```

Соответственно, это заглушка и скелет. После их создания можно получить сервер и клиент для общения между собой.

Главная цель RMI состоит в упрощении использования удаленных объектов. Есть только одна дополнительная вещь, которую необходимо выполнить в клиентской программе, это найти и получить удаленный интерфейс с сервера. Все остальное – это обычное Java-программирование: посылка сообщений объекту. Ниже приведен пример программы, которая использует PerfectTime:

```
// Использование удаленного объекта PerfectTime.
// {Broken}
package c15.rmi;
import java.rmi.*;
import java.rmi.registry.*;
public class DisplayPerfectTime {
```

```
public static void main(String[] args) throws Exception {
    System.setSecurityManager(new RMISecurityManager());
    PerfectTimeI t = (PerfectTimeI) Naming
          .lookup("//peppy:2005/PerfectTime");
    for (int i = 0; i < 10; i++)
     System.out.println("Perfect time = " +t.getPerfectTime());
     }
       } //
```

Строка идентификатора точно такая же, как и используемая для регистрации объекта с помощью Naming, а первая часть представляет URL и номер порта. Так как используется URL, то можно также указать машину в Internet'e.

То, что приходит от Naming.lookup( ), должно быть приведено к удаленному интерфейсу, а не к классу. Если вместо этого будет использоваться класс, то получится исключение. Можно видеть вызов метода t.getPerfectTime(), т. к. то, что есть в наличии, – это ссылка на удаленный объект, программирование с которой не отличается от программирования с локальным объектом (с одним отличием: удаленные методы выбрасывают RemoteException).

**Задание**

Создать систему, функционально состоящую из двух компонентов – сервера (процессингового центра) и клиента (касса). Предполагается, что сервер в системе один (именно он обладает всей информацией о зарегистрированных картах и их балансах), а касс много и на них проходят операции регистрации новых карт, а также операции изменения баланса карт (соответственно – оплаты и занесения наличных).

***Контрольные вопросы***

1 Поясните назначение RMI.

2 Перечислите особенности создания удаленного интерфейса.

3 Приведите пример реализации удаленного интерфейса.

4 Перечислите действия, которые необходимо выполнить для обслуживания RMI-объектов.

5 Особенности работы с RMI.

# Список литературы

1 Программирование сетевых приложений : лабораторные работы (практикум) для студентов специальностей 1-40 01 01 «Программное обеспечение информационных технологий» и 1-40 01 02 «Информационные системы и технологии» / сост. Н. М. Прибыльская. – Мн.: БНТУ, 2013. – 67 с.

2 **Флэнаган, Д.** JavaScript : карманный справочник : пер. с англ. / Д. Флэнаган. – 3-е изд. – М. : Вильямс, 2015. – 314 с.

3 **Шилдт, Г.** Java 8. Полное руководство : пер. с англ. / Г. Шилдт. – 9-е изд. – М. : Вильямс, 2015. – 1375 с.

4 **Курняван, Б.** Программирование WEB-приложений на языке Java : пер. с англ. / Б. Курняван. – М. : Лори, 2014. – 880 с.